CPSC 330
Applied Machine Learning

Lecture 10: Regression Evaluation Metrics

UBC 2022-23

Instructor: Mathias Lécuyer

Announcement

- Midterm next week, on Wednseday, Feb. 15, from 7:00pm to 8:15pm:
  - CPSC 330 Section 202 students (you) will write the exam in ESB 1012 (https://learningspaces.ubc.ca/classrooms/esb-1012 (https://learningspaces.ubc.ca/classrooms/esb-1012)):
  - More details on piazza (https://piazza.com/class/lcgo6c2ncl06el/post/283): https://piazza.com/class/lcgo6c2ncl06el/post/283 (https://piazza.com/class/lcgo6c2ncl06el/post/283).
- There is a piazza poll (https://piazza.com/class/lcgo6c2ncl06el/post/316) for topics to cover in the review session (next Tuesday, Feb 14): https://piazza.com/class/lcgo6c2ncl06el/post/316 (https://piazza.com/class/lcgo6c2ncl06el/post/316)

Processing math: 100%

Imports

In [41]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.compose import (
    ColumnTransformer,
    TransformedTargetRegressor,
    make_column_transformer,
)
from sklearn.dummy import DummyRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV
from sklearn.metrics import make_scorer, mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score, cross_validate, tr
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, Standa
from sklearn.tree import DecisionTreeRegressor

%matplotlib inline
```

In [42]:
```
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
```

Some clarifications on last lecture

- Formula for false positive rate: Fraction of false positives out of all negative examples:

$$FPR = \frac{FP}{FP + TN} = \frac{FP}{N}$$

- This was in context of the ROC curve, which is TPR (a.k.a recall, a.k.a sensitivity) as function of FPR:

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{P}, \; FPR = \frac{FP}{FP + TN}$$

- Recall is also called sensitivity:

$$\textrm{Recall} = \textrm{Sensitivity} = \frac{TP}{TP + FN} = \frac{TP}{P}$$

- Note that TPR/recall/sensitivity is about the positive class (how much of it we find), while FPR iss really about the negative class (how much of it we mispredict).
- Precision, recall, f1 score, are only about the positive label:

$$\textrm{Precision} = \frac{TP}{TP + FP}, \ \textrm{Recall} = \textrm{Sensitivity} = \frac{TP}{TP + FN} = \frac{TP}{P}$$

Processing math: 100%

- The positive class is assumed to be the class label 1 by default. This is configurable through the `pos_label` parameter.

Learning outcomes

From this lecture, students are expected to be able to:

- Carry out feature transformations on somewhat complicated dataset.
- Visualize transformed features as a dataframe.
- Use `Ridge` and `RidgeCV`.
- Explain how `alpha` hyperparameter of `Ridge` relates to the fundamental tradeoff.
- Examine coefficients of transformed features.
- Appropriately select a scoring metric given a regression problem.
- Interpret and communicate the meanings of different scoring metrics on regression problems.
  - MSE, RMSE, R^2, MAPE
- Apply log-transform on the target values in a regression problem with `TransformedTargetRegressor`.

Dataset

In this lecture, we'll be using [Kaggle House Prices dataset (https://www.kaggle.com/c/home-data-for-ml-course/)](https://www.kaggle.com/c/home-data-for-ml-course/). As usual, to run this notebook you'll need to download the data. For this dataset, train and test have already been separated. We'll be working with the train portion in this lecture.

In [43]:

```
df = pd.read_csv("../data/housing-kaggle/train.csv")
train_df, test_df = train_test_split(df, test_size=0.10, random_state=1
train_df.head()
```

Out[43]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	L
302	303	20	RL	118.0	13704	Pave	NaN	IR1	Lvl	
767	768	50	RL	75.0	12508	Pave	NaN	IR1	Lvl	
429	430	20	RL	130.0	11457	Pave	NaN	IR1	Lvl	
1139	1140	30	RL	98.0	8731	Pave	NaN	IR1	Lvl	
558	559	60	RL	57.0	21872	Pave	NaN	IR2	HLS	

5 rows × 81 columns

- The supervised machine learning problem is predicting housing price given features associated with properties.
- Here, the target is `SalePrice`, which is continuous. So it's a **regression problem** (as opposed to classification).

Processing math: 100%

In [44]:
```
train_df.shape
```

Out[44]: (1314, 81)

Let's separate `X` and `y`

In [45]:
```
X_train = train_df.drop(columns=["SalePrice"])
y_train = train_df["SalePrice"]

X_test = test_df.drop(columns=["SalePrice"])
y_test = test_df["SalePrice"]
```

EDA

In [46]:
```
train_df.describe()
```

Out[46]:

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBui
count	1314.000000	1314.000000	1089.000000	1314.000000	1314.000000	1314.000000	1314.00000
mean	734.182648	56.472603	69.641873	10273.261035	6.076104	5.570015	1970.99543
std	422.224662	42.036646	23.031794	8997.895541	1.392612	1.112848	30.19812
min	1.000000	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.00000
25%	369.250000	20.000000	59.000000	7500.000000	5.000000	5.000000	1953.00000
50%	735.500000	50.000000	69.000000	9391.000000	6.000000	5.000000	1972.00000
75%	1099.750000	70.000000	80.000000	11509.000000	7.000000	6.000000	2000.00000
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.00000

8 rows × 38 columns

Processing math: 100%

In [47]:
```
train_df.info()
```

Processing math: 100%

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1314 entries, 302 to 1389
Data columns (total 81 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             1314 non-null   int64
 1   MSSubClass     1314 non-null   int64
 2   MSZoning       1314 non-null   object
 3   LotFrontage    1089 non-null   float64
 4   LotArea        1314 non-null   int64
 5   Street         1314 non-null   object
 6   Alley          81 non-null     object
 7   LotShape       1314 non-null   object
 8   LandContour    1314 non-null   object
 9   Utilities      1314 non-null   object
 10  LotConfig      1314 non-null   object
 11  LandSlope      1314 non-null   object
 12  Neighborhood   1314 non-null   object
 13  Condition1     1314 non-null   object
 14  Condition2     1314 non-null   object
 15  BldgType       1314 non-null   object
 16  HouseStyle     1314 non-null   object
 17  OverallQual    1314 non-null   int64
 18  OverallCond    1314 non-null   int64
 19  YearBuilt      1314 non-null   int64
 20  YearRemodAdd   1314 non-null   int64
 21  RoofStyle      1314 non-null   object
 22  RoofMatl       1314 non-null   object
 23  Exterior1st    1314 non-null   object
 24  Exterior2nd    1314 non-null   object
 25  MasVnrType     1307 non-null   object
 26  MasVnrArea     1307 non-null   float64
 27  ExterQual      1314 non-null   object
 28  ExterCond      1314 non-null   object
 29  Foundation     1314 non-null   object
 30  BsmtQual       1280 non-null   object
 31  BsmtCond       1280 non-null   object
 32  BsmtExposure   1279 non-null   object
 33  BsmtFinType1   1280 non-null   object
 34  BsmtFinSF1     1314 non-null   int64
 35  BsmtFinType2   1280 non-null   object
 36  BsmtFinSF2     1314 non-null   int64
 37  BsmtUnfSF      1314 non-null   int64
 38  TotalBsmtSF    1314 non-null   int64
 39  Heating        1314 non-null   object
 40  HeatingQC      1314 non-null   object
 41  CentralAir     1314 non-null   object
 42  Electrical     1313 non-null   object
 43  1stFlrSF       1314 non-null   int64
 44  2ndFlrSF       1314 non-null   int64
 45  LowQualFinSF   1314 non-null   int64
 46  GrLivArea      1314 non-null   int64
 47  BsmtFullBath   1314 non-null   int64
 48  BsmtHalfBath   1314 non-null   int64
 49  FullBath       1314 non-null   int64
 50  HalfBath       1314 non-null   int64
 51  BedroomAbvGr   1314 non-null   int64
```

Processing math: 100%

```
 52  KitchenAbvGr   1314 non-null   int64
 53  KitchenQual    1314 non-null   object
 54  TotRmsAbvGrd   1314 non-null   int64
 55  Functional     1314 non-null   object
 56  Fireplaces     1314 non-null   int64
 57  FireplaceQu    687 non-null    object
 58  GarageType     1241 non-null   object
 59  GarageYrBlt    1241 non-null   float64
 60  GarageFinish   1241 non-null   object
 61  GarageCars     1314 non-null   int64
 62  GarageArea     1314 non-null   int64
 63  GarageQual     1241 non-null   object
 64  GarageCond     1241 non-null   object
 65  PavedDrive     1314 non-null   object
 66  WoodDeckSF     1314 non-null   int64
 67  OpenPorchSF    1314 non-null   int64
 68  EnclosedPorch  1314 non-null   int64
 69  3SsnPorch      1314 non-null   int64
 70  ScreenPorch    1314 non-null   int64
 71  PoolArea       1314 non-null   int64
 72  PoolQC         7 non-null      object
 73  Fence          259 non-null    object
 74  MiscFeature    50 non-null     object
 75  MiscVal        1314 non-null   int64
 76  MoSold         1314 non-null   int64
 77  YrSold         1314 non-null   int64
 78  SaleType       1314 non-null   object
 79  SaleCondition  1314 non-null   object
 80  SalePrice      1314 non-null   int64
dtypes: float64(3), int64(35), object(43)
memory usage: 841.8+ KB
```

pandas_profiler

We do not have `pandas_profiling` in our course environment. You will have to install it in the environment on your own if you want to run the code below.

```
conda install -c conda-forge pandas-profiling
```

In [48]:

```python
from pandas_profiling import ProfileReport

#profile = ProfileReport(train_df, title="Pandas Profiling Report")  #
#profile.to_notebook_iframe()
```

Processing math: 100%

Feature types

- Do not blindly trust all the info given to you by automated tools.
- How does pandas profiling figure out the data type?
  - You can look at the Python data type and say floats are numeric, strings are categorical.
  - However, in doing so you would miss out on various subtleties such as some of the string features being ordinal rather than truly categorical.
  - Also, it will think free text is categorical.

- In addition to tools such as above, it's important to go through data description to understand the data.
- The data description for our dataset is available [here (https://www.kaggle.com/c/home-data-for-ml-course/data?select=data_description.txt)](https://www.kaggle.com/c/home-data-for-ml-course/data?select=data_description.txt).

Feature types

- We have mixed feature types and a bunch of missing values.
- Now, let's identify feature types and transformations.

- Let's get the numeric-looking columns.

In [49]:
```
numeric_looking_columns = X_train.select_dtypes(include=np.number).colu
print(numeric_looking_columns)
```

```
['Id', 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCon
d', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF
2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF',
'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'Bed
roomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt',
'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch',
'3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold']
```

Not all numeric looking columns are necessarily numeric.

In [50]:
```
train_df["MSSubClass"].unique()
```

Out[50]:
```
array([ 20,  50,  30,  60, 160,  85,  90, 120, 180,  80,  70,  75, 190,
        45,  40])
```

MSSubClass: Identifies the type of dwelling involved in the sale.

Processing math: 100%

```
        20  1-STORY 1946 & NEWER ALL STYLES
        30  1-STORY 1945 & OLDER
        40  1-STORY W/FINISHED ATTIC ALL AGES
        45  1-1/2 STORY - UNFINISHED ALL AGES
        50  1-1/2 STORY FINISHED ALL AGES
        60  2-STORY 1946 & NEWER
        70  2-STORY 1945 & OLDER
        75  2-1/2 STORY ALL AGES
        80  SPLIT OR MULTI-LEVEL
        85  SPLIT FOYER
        90  DUPLEX - ALL STYLES AND AGES
       120  1-STORY PUD (Planned Unit Development) - 1946 & NEWER
```

Also, month sold is more of a categorical feature than a numeric feature.

In [51]:
```
train_df["MoSold"].unique() # Month Sold
```

Out[51]:
```
array([ 1,  7,  3,  5,  8, 10,  6,  9, 12,  2,  4, 11])
```

Processing math: 100%

In [52]:
```
drop_features = ["Id"]
numeric_features = [
    "BedroomAbvGr",
    "KitchenAbvGr",
    "LotFrontage",
    "LotArea",
    "OverallQual",
    "OverallCond",
    "YearBuilt",
    "YearRemodAdd",
    "MasVnrArea",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtUnfSF",
    "TotalBsmtSF",
    "1stFlrSF",
    "2ndFlrSF",
    "LowQualFinSF",
    "GrLivArea",
    "BsmtFullBath",
    "BsmtHalfBath",
    "FullBath",
    "HalfBath",
    "TotRmsAbvGrd",
    "Fireplaces",
    "GarageYrBlt",
    "GarageCars",
    "GarageArea",
    "WoodDeckSF",
    "OpenPorchSF",
    "EnclosedPorch",
    "3SsnPorch",
    "ScreenPorch",
    "PoolArea",
    "MiscVal",
    "YrSold",
]
```

```
I've not looked at all the features carefully. It might be appropri
ate to apply some other encoding on some of the numeric features ab
ove.
```

In [53]:
```
set(numeric_looking_columns) - set(numeric_features) - set(drop_feature
```

Out[53]:
```
{'MSSubClass', 'MoSold'}
```

We'll treat the above numeric-looking features as categorical features.

- There are a bunch of ordinal features in this dataset.
- Ordinal features with the same scale
  - Poor (Po), Fair (Fa), Typical (TA), Good (Gd), Excellent (Ex)

Processing math: 100%

- These we'll be calling `ordinal_features_reg`.
- Ordinal features with different scales
  - These we'll be calling `ordinal_features_oth`.

In [54]:
```python
ordinal_features_reg = [
    "ExterQual",
    "ExterCond",
    "BsmtQual",
    "BsmtCond",
    "HeatingQC",
    "KitchenQual",
    "FireplaceQu",
    "GarageQual",
    "GarageCond",
    "PoolQC",
]
ordering = [
    "Po",
    "Fa",
    "TA",
    "Gd",
    "Ex",
]  # if N/A it will just impute something, per below
ordering_ordinal_reg = [ordering] * len(ordinal_features_reg)
ordering_ordinal_reg
```

Out[54]:
```
[['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex']]
```

We'll pass the above as categories in our `OrdinalEncoder`.

- There are a bunch more ordinal features using different scales.
  - These we'll be calling `ordinal_features_oth`.
  - We are encoding them separately.

Processing math: 100%

In [55]:
```
ordinal_features_oth = [
    "BsmtExposure",
    "BsmtFinType1",
    "BsmtFinType2",
    "Functional",
    "Fence",
]
ordering_ordinal_oth = [
    ['NA', 'No', 'Mn', 'Av', 'Gd'],
    ['NA', 'Unf', 'LwQ', 'Rec', 'BLQ', 'ALQ', 'GLQ'],
    ['NA', 'Unf', 'LwQ', 'Rec', 'BLQ', 'ALQ', 'GLQ'],
    ['Sal','Sev','Maj2','Maj1','Mod','Min2','Min1','Typ'],
    ['NA', 'MnWw', 'GdWo', 'MnPrv', 'GdPrv']
]
```

The remaining features are categorical features.

Processing math: 100%

In [56]:
```
categorical_features = list(
    set(X_train.columns)
    - set(numeric_features)
    - set(ordinal_features_reg)
    - set(ordinal_features_oth)
    - set(drop_features)
)
categorical_features
```

Out[56]:
```
['MasVnrType',
 'Neighborhood',
 'Condition2',
 'Alley',
 'SaleCondition',
 'Electrical',
 'HouseStyle',
 'GarageFinish',
 'RoofStyle',
 'MoSold',
 'Exterior1st',
 'Street',
 'MSSubClass',
 'LotShape',
 'PavedDrive',
 'Heating',
 'SaleType',
 'Utilities',
 'GarageType',
 'BldgType',
 'MiscFeature',
 'LotConfig',
 'CentralAir',
 'LandSlope',
 'Condition1',
 'Exterior2nd',
 'MSZoning',
 'LandContour',
 'RoofMatl',
 'Foundation']
```

- We are not doing it here but we can engineer our own features too.
- Would price per square foot be a good feature to add in here?

Applying feature transformations

- Since we have mixed feature types, let's use `ColumnTransformer` to apply different transformations on different features types.

Processing math: 100%

In [57]:
```
from sklearn.compose import ColumnTransformer, make_column_transformer

numeric_transformer = make_pipeline(SimpleImputer(strategy="median"), S
ordinal_transformer_reg = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(categories=ordering_ordinal_reg),
)

ordinal_transformer_oth = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(categories=ordering_ordinal_oth),
)

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore", sparse=False),
)

preprocessor = make_column_transformer(
    ("drop", drop_features),
    (numeric_transformer, numeric_features),
    (ordinal_transformer_reg, ordinal_features_reg),
    (ordinal_transformer_oth, ordinal_features_oth),
    (categorical_transformer, categorical_features),
)
```

Processing math: 100%

Examining the preprocessed data

In [58]:
```
preprocessor.fit(X_train) # Calling fit to examine all the transformers
preprocessor.named_transformers_
```

Processing math: 100%

Out[58]:
```
{'drop': 'drop',
 'pipeline-1': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy
='median')),
                 ('standardscaler', StandardScaler())]),
 'pipeline-2': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy
='most_frequent')),
                 ('ordinalencoder',
                  OrdinalEncoder(categories=[['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x']]))]),
 'pipeline-3': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy
='most_frequent')),
                 ('ordinalencoder',
                  OrdinalEncoder(categories=[['NA', 'No', 'Mn', 'Av', 'G
d'],
                                             ['NA', 'Unf', 'LwQ', 'Rec',
'BLQ',
                                              'ALQ', 'GLQ'],
                                             ['NA', 'Unf', 'LwQ', 'Rec',
'BLQ',
                                              'ALQ', 'GLQ'],
                                             ['Sal', 'Sev', 'Maj2', 'Maj
1',
                                              'Mod', 'Min2', 'Min1', 'Ty
p'],
                                             ['NA', 'MnWw', 'GdWo', 'MnPr
v',
                                              'GdPrv']]))]),
 'pipeline-4': Pipeline(steps=[('simpleimputer',
                  SimpleImputer(fill_value='missing', strategy='constan
t')),
                 ('onehotencoder',
                  OneHotEncoder(handle_unknown='ignore', sparse=Fals
e))])}
```

Processing math: 100%

In [59]:
```
ohe_columns = list(
    preprocessor.named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names(categorical_features)
)
new_columns = numeric_features + ordinal_features_reg + ordinal_feature
```

In [60]:
```
X_train_enc = pd.DataFrame(
    preprocessor.transform(X_train), index=X_train.index, columns=new_c
)
X_train_enc.head()
```

Out[60]:

	BedroomAbvGr	KitchenAbvGr	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	Ye
302	0.154795	-0.222647	2.312501	0.381428	0.663680	-0.512408	0.993969	
767	1.372763	-0.222647	0.260890	0.248457	-0.054669	1.285467	-1.026793	
429	0.154795	-0.222647	2.885044	0.131607	-0.054669	-0.512408	0.563314	
1139	0.154795	-0.222647	1.358264	-0.171468	-0.773017	-0.512408	-1.689338	
558	0.154795	-0.222647	-0.597924	1.289541	0.663680	-0.512408	0.828332	

5 rows × 263 columns

In [61]:
```
X_train.shape
```

Out[61]: (1314, 80)

In [62]:
```
X_train_enc.shape
```

Out[62]: (1314, 263)

We went from 80 features to 263 features!!

Other possible preprocessing?

- There is a lot of room for improvement ...
- We're just using `SimpleImputer`.
  - In reality we'd want to go through this more carefully.
  - We may also want to drop some columns that are almost entirely missing.
- We could also check for outliers, and do other exploratory data analysis (EDA).
- But for now this is good enough ...

Processing math: 100%

Model building

`DummyRegressor`

In [63]:

```
dummy = DummyRegressor()
pd.DataFrame(cross_validate(dummy, X_train, y_train, cv=10, return_trai
```

Out[63]:

	fit_time	score_time	test_score	train_score
0	0.001767	0.000557	-0.003547	0.0
1	0.001569	0.000441	-0.001266	0.0
2	0.001287	0.000450	-0.011767	0.0
3	0.002081	0.000792	-0.006744	0.0
4	0.001895	0.000488	-0.076533	0.0
5	0.001080	0.000369	-0.003133	0.0
6	0.001010	0.000363	-0.000397	0.0
7	0.000984	0.000360	-0.003785	0.0
8	0.001093	0.000494	-0.001740	0.0
9	0.001242	0.000619	-0.000117	0.0

Apply `Ridge`

- Recall that we are going to use `Ridge()` instead of `LinearRegression()` in this course.
  - It has a hyperparameter `alpha` which controls the fundamental tradeoff.

In [64]:

```
lr_pipe = make_pipeline(preprocessor, Ridge())
pd.DataFrame(cross_validate(lr_pipe, X_train, y_train, cv=10, return_tr
```

Out[64]:

	fit_time	score_time	test_score	train_score
0	0.067918	0.020969	0.861355	0.911906
1	0.073782	0.020503	0.812301	0.913861
2	0.075481	0.021917	0.775283	0.915963
3	0.077557	0.021604	0.874519	0.910849
4	0.074957	0.021669	0.851969	0.911622
5	0.074832	0.024669	0.826198	0.910176
6	0.074639	0.022231	0.825533	0.913781
7	0.074474	0.021569	0.872238	0.910071
8	0.077648	0.024597	0.196663	0.921448
9	0.072799	0.023196	0.890474	0.908221

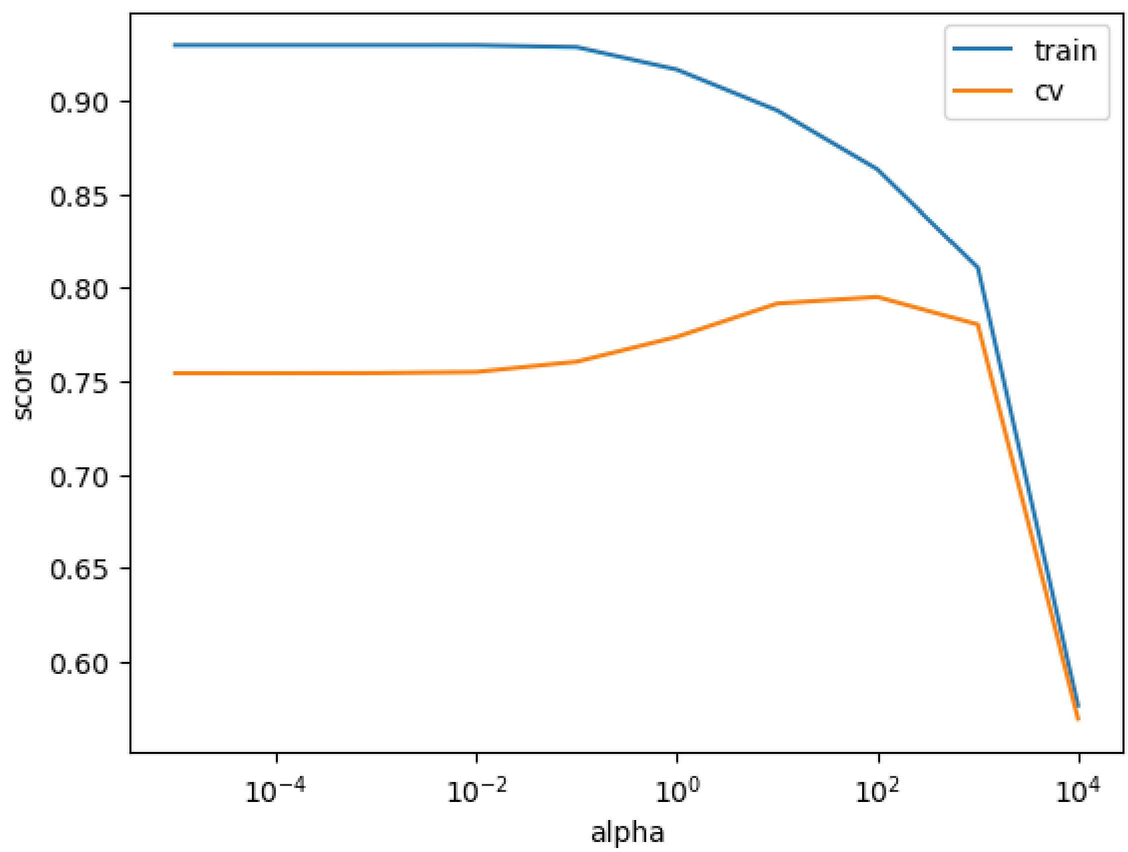
- Quite a bit of variation in the test scores.
- Performing poorly in fold 8. Not sure why.

Tuning the `alpha` hyperparameter of `Ridge`

- Recall that `Ridge` has a hyperparameter `alpha` that controls the fundamental tradeoff.
- This is like `C` in `LogisticRegression` but, annoyingly, `alpha` is the opposite of `C` : large `C` is like small `alpha` and vice versa.
- Smaller `alpha` : more complex model, more variance, lower training error (overfitting)

In [65]:
```
alphas = 10.0 ** np.arange(-5, 5, 1)
train_scores = []
cv_scores = []
for alpha in alphas:
    lr = make_pipeline(preprocessor, Ridge(alpha=alpha))
    results = cross_validate(lr, X_train, y_train, return_train_score=T
    train_scores.append(np.mean(results["train_score"]))
    cv_scores.append(np.mean(results["test_score"]))
```

In [66]:
```
plt.semilogx(alphas, train_scores, label="train")
plt.semilogx(alphas, cv_scores, label="cv")
plt.legend()
plt.xlabel("alpha")
plt.ylabel("score");
```



Processing math: 100%

In [67]:
```
best_alpha = alphas[np.argmax(cv_scores)]
best_alpha
```

Out[67]: 100.0

- It seems alpha=100 is the best choice here.

- General intuition: larger `alpha` leads to smaller coefficients.
- Smaller coefficients mean the predictions are less sensitive to changes in the data.
- Hence less chance of overfitting (seeing big dependencies when you shouldn't).

`RidgeCV`

BTW, because it's so common to want to tune `alpha` with `Ridge` , sklearn provides a class called `RidgeCV` , which automatically tunes `alpha` based on cross-validation.

In [68]:
```
ridgecv_pipe = make_pipeline(preprocessor, RidgeCV(alphas=alphas, cv=10
ridgecv_pipe.fit(X_train, y_train);
```

In [69]:
```
best_alpha = ridgecv_pipe.named_steps['ridgecv'].alpha_
best_alpha
```

Out[69]: 100.0

Let's examine the coefficients

In [70]:
```
lr_tuned = make_pipeline(preprocessor, Ridge(alpha=best_alpha))
lr_tuned.fit(X_train, y_train)
lr_preds = lr_tuned.predict(X_test)
lr_preds[:10]
```

Out[70]:
```
array([228728.1963872 , 104718.39905565, 155778.96723311, 246316.7111903
1,
       127633.10676873, 243207.19441128, 304930.24461291, 145374.5943529
5,
       157059.38983893, 128487.51979632])
```

In [71]:
```
lr_preds.max(), lr_preds.min()
```

Out[71]: (390726.10647423274, 30791.092505420726)

Let's get the feature names of the transformed data to interpret coefficients.

Processing math: 100%

In [72]:

```
ohe_columns = list(
    preprocessor.named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names(categorical_features)
)
new_columns = numeric_features + ordinal_features_reg + ordinal_feature

```

In [73]:

```
df = pd.DataFrame(
    data={
        "features": new_columns,
        "coefficients": lr_tuned.named_steps["ridge"].coef_,
    }
)
```

In [74]:

```
df.sort_values("coefficients",ascending=False)
```

Out[74]:

	features	coefficients
4	OverallQual	14484.902165
16	GrLivArea	11704.053037
70	Neighborhood_NridgHt	9662.969631
69	Neighborhood_NoRidge	9497.598615
36	BsmtQual	8073.088562
...	...	...
249	RoofMatl_ClyTile	-3992.399179
245	LandContour_Bnk	-5001.996997
62	Neighborhood_Gilbert	-5197.585536
59	Neighborhood_CollgCr	-5467.463086
61	Neighborhood_Edwards	-5796.508529

263 rows × 2 columns

So according to this model:

- As `OverallQual` feature gets bigger the housing price will get bigger.
- Presence of `Neighborhood_Edwards` will result in smaller median house value.

Processing math: 100%

In [75]:
```
X_train_enc['Neighborhood_Edwards']
```

Out[75]:
```
302     0.0
767     0.0
429     0.0
1139    0.0
558     0.0
       ...
1041    0.0
1122    1.0
1346    0.0
1406    0.0
1389    0.0
Name: Neighborhood_Edwards, Length: 1314, dtype: float64
```

Regression score functions

- We aren't doing classification anymore, so we can't just check for equality:

In [76]:
```
# This doesn't make sense:
lr_tuned.predict(X_train) == y_train
```

Out[76]:
```
302     False
767     False
429     False
1139    False
558     False
        ...
1041    False
1122    False
1346    False
1406    False
1389    False
Name: SalePrice, Length: 1314, dtype: bool
```

In [77]:
```
y_train.values
```

Out[77]:
```
array([205000, 160000, 175000, ..., 262500, 133000, 131000])
```

In [78]:
```
lr_tuned.predict(X_train)
```

Out[78]:
```
array([212894.62756285, 178502.78223444, 189937.18327372, ...,
       245233.6751565 , 129863.13373552, 135439.89186716])
```

Processing math: 100%

We need a score that reflects how right/wrong each prediction is.

There are a number of popular scoring functions for regression. We are going to look at some common metrics:

- mean squared error (MSE)
- R^2
- root mean squared error (RMSE)
- MAPE

See [sklearn documentation (https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics)](https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics) for more details.

Mean squared error (MSE)

- A common metric is mean squared error:

In [79]:
```
preds = lr_tuned.predict(X_train)
```

In [80]:
```
np.mean((y_train - preds) ** 2)
```

Out[80]:
```
873230473.3636098
```

Perfect predictions would have MSE=0.

In [81]:
```
np.mean((y_train - y_train) ** 2)
```

Out[81]:
```
0.0
```

This is also implemented in sklearn:

In [82]:
```
from sklearn.metrics import mean_squared_error

mean_squared_error(y_train, preds)
```

Out[82]:
```
873230473.3636098
```

- MSE looks huge and unreasonable. There is an error of ~$1 Billion!
- Is this score good or bad?

- Unlike classification, with regression **our target has units.**
- The target is in dollars, the mean squared error is in $dollars^2$
- The score also depends on the scale of the targets.
- If we were working in cents instead of dollars, our MSE would be 10,000 times (100^2) higher!

Processing math: 100%

In [83]:
```
np.mean((y_train * 100 - preds * 100) ** 2)
```
Out[83]: 8732304733636.098

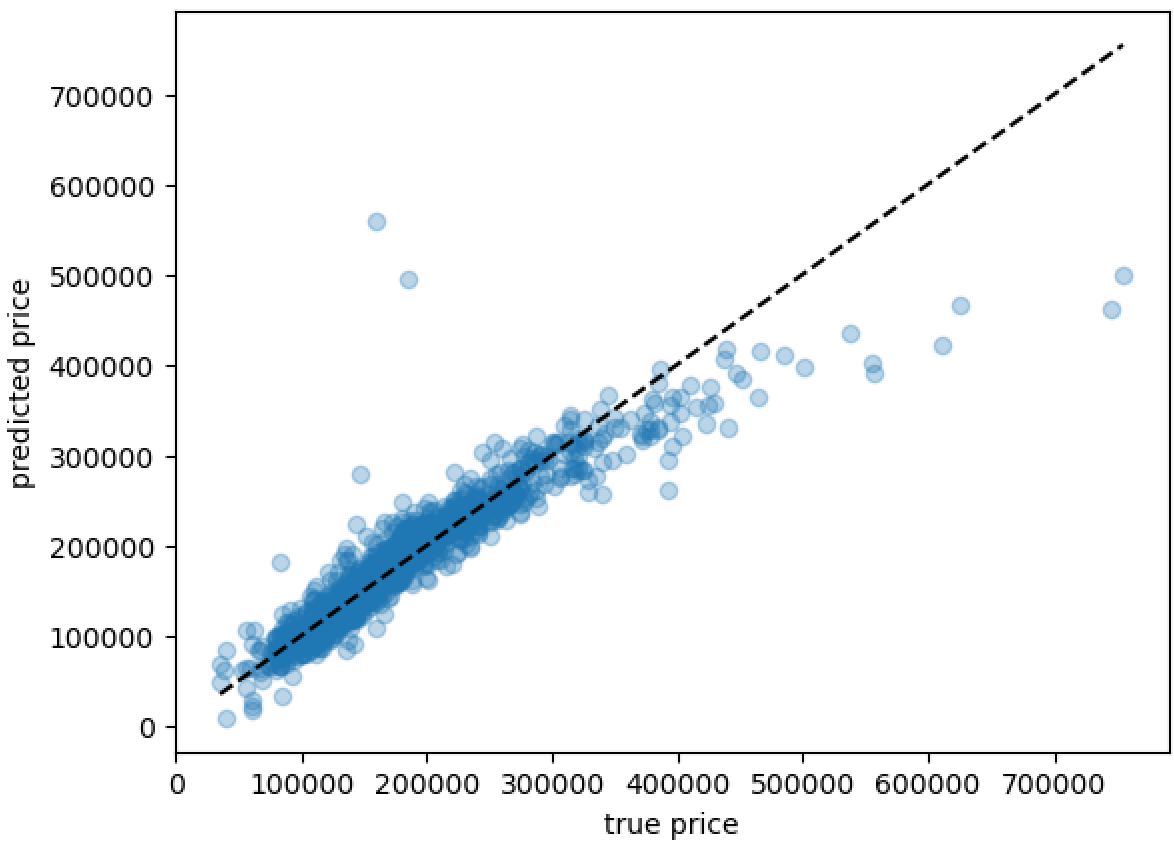
Root mean squared error or RMSE

- The MSE above is in $dollars^2$.
- A more relatable metric would be the root mean squared error, or RMSE

In [84]:
```
np.sqrt(mean_squared_error(y_train, lr_tuned.predict(X_train)))
```
Out[84]: 29550.473318774606

- Error of $30,000 makes more sense.
- Can we dig deeper?

In [85]:
```
plt.scatter(y_train, lr_tuned.predict(X_train), alpha=0.3)
grid = np.linspace(y_train.min(), y_train.max(), 1000)
plt.plot(grid, grid, "--k")
plt.xlabel("true price")
plt.ylabel("predicted price");
```



- Here we can see a few cases where our prediction is way off.
- Is there something weird about those houses, perhaps? Outliers?
- Under the line means we're under-prediction, over the line means we're over-predicting.

Processing math: 100%

R^2 (not in detail)

A common score is the R^2

- This is the score that `sklearn` uses by default when you call score():
- You can [read about it (https://en.wikipedia.org/wiki/Coefficient_of_determination)](https://en.wikipedia.org/wiki/Coefficient_of_determination) if interested.
- Intuition: similar to mean squared error, but flipped (higher is better), and normalized so the max is 1.

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y_i})^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Key points:

- The maximum is 1 for perfect predictions
- Negative values are very bad: "worse than DummyRegressor" (very bad)

(optional) Warning: MSE is "reversible" but R^2 is not:

In [86]:
```
mean_squared_error(y_train, preds)
```
Out[86]: 873230473.3636098

In [87]:
```
mean_squared_error(preds, y_train)
```
Out[87]: 873230473.3636098

In [88]:
```
r2_score(y_train, preds)
```
Out[88]: 0.8601212294857903

In [89]:
```
r2_score(preds, y_train)
```
Out[89]: 0.827962225882707

- When you call `fit` it minimizes MSE / maximizes R^2 (or something like that) by default.
- Just like in classification, this isn't always what you want!!

MAPE

- We got an RMSE of ~$30,000 before.

Question: Is an error of $30,000 acceptable?

Processing math: 100%

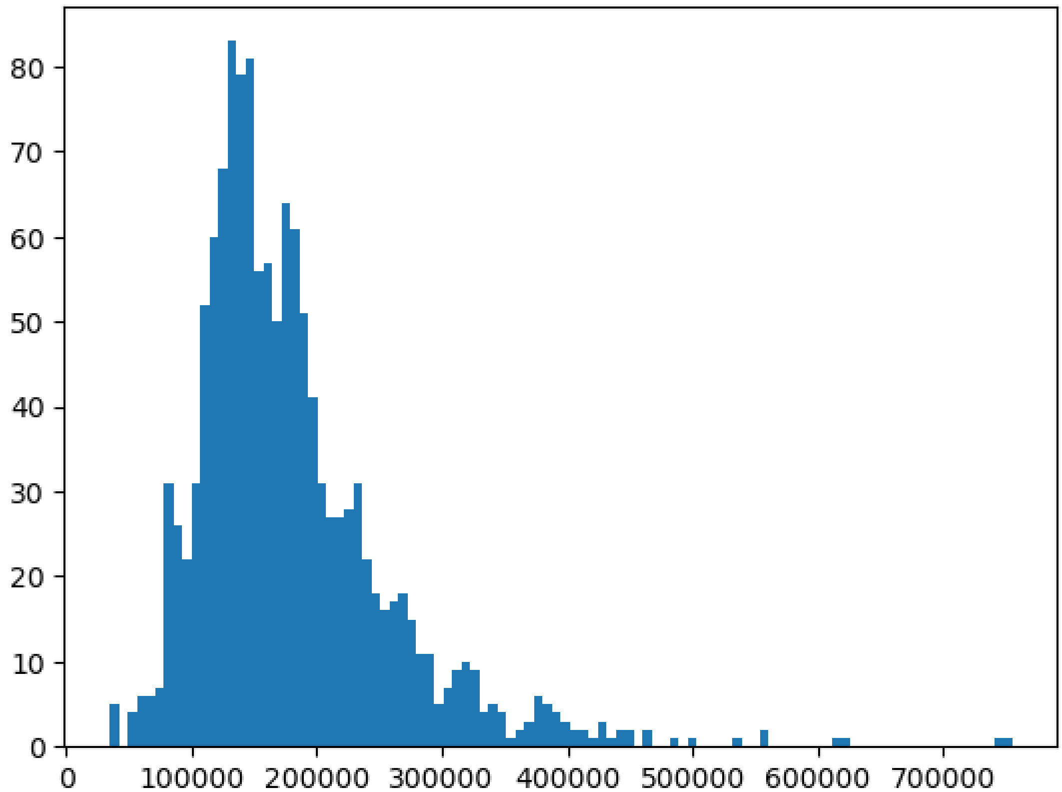
In [90]:
```
np.sqrt(mean_squared_error(y_train, lr_tuned.predict(X_train)))
```

Out[90]: 29550.473318774606

- For a house worth $600k, it seems reasonable! That's 5% error.
- For a house worth $60k, that is terrible. It's 50% error.

We have both of these cases in our dataset.

In [91]:
```
plt.hist(y_train, bins=100);
```



How about looking at percent error?

Processing math: 100%

In [92]:

```
pred_train = lr_tuned.predict(X_train)
percent_errors = (pred_train - y_train) / y_train * 100.0
percent_errors
```

Out[92]:

```
302      3.851038
767     11.564239
429      8.535533
1139   -16.371069
558     17.177968
          ...
1041    -0.496571
1122   -28.696351
1346    -6.577648
1406    -2.358546
1389     3.389230
Name: SalePrice, Length: 1314, dtype: float64
```

These are both positive (predict too high) and negative (predict too low).

We can look at the absolute percent error:

In [93]:

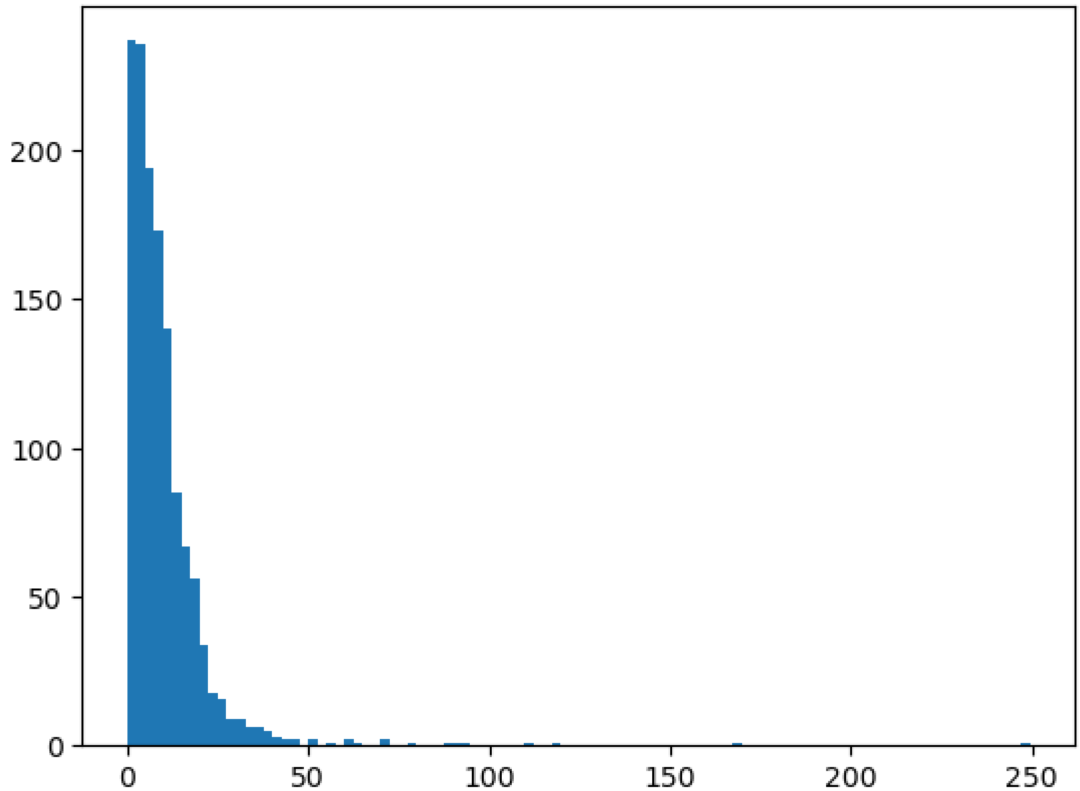
```
np.abs(percent_errors)
```

Out[93]:

```
302      3.851038
767     11.564239
429      8.535533
1139    16.371069
558     17.177968
          ...
1041     0.496571
1122    28.696351
1346     6.577648
1406     2.358546
1389     3.389230
Name: SalePrice, Length: 1314, dtype: float64
```

Processing math: 100%

In [94]:
```
plt.hist(np.abs(percent_errors), bins=100);
```



And, like MSE, we can take the average over examples. This is called mean absolute percent error (MAPE).

In [95]:
```
def mape(true, pred):
    return 100.0 * np.mean(np.abs((pred - true) / true))
```

In [96]:
```
mape(y_train, pred_train)
```

Out[96]: 10.093121294225265

- Ok, this is quite interpretable.
- On average, we have around 10% error.

Transforming the targets

- Does `.fit()` know we care about MAPE?
- No, it doesn't. Why are we minimizing MSE (or something similar) if we care about MAPE??
- When minimizing MSE, the expensive houses will dominate because they have the biggest error.
- Which is better for RMSE?

Processing math: 100%

Model A

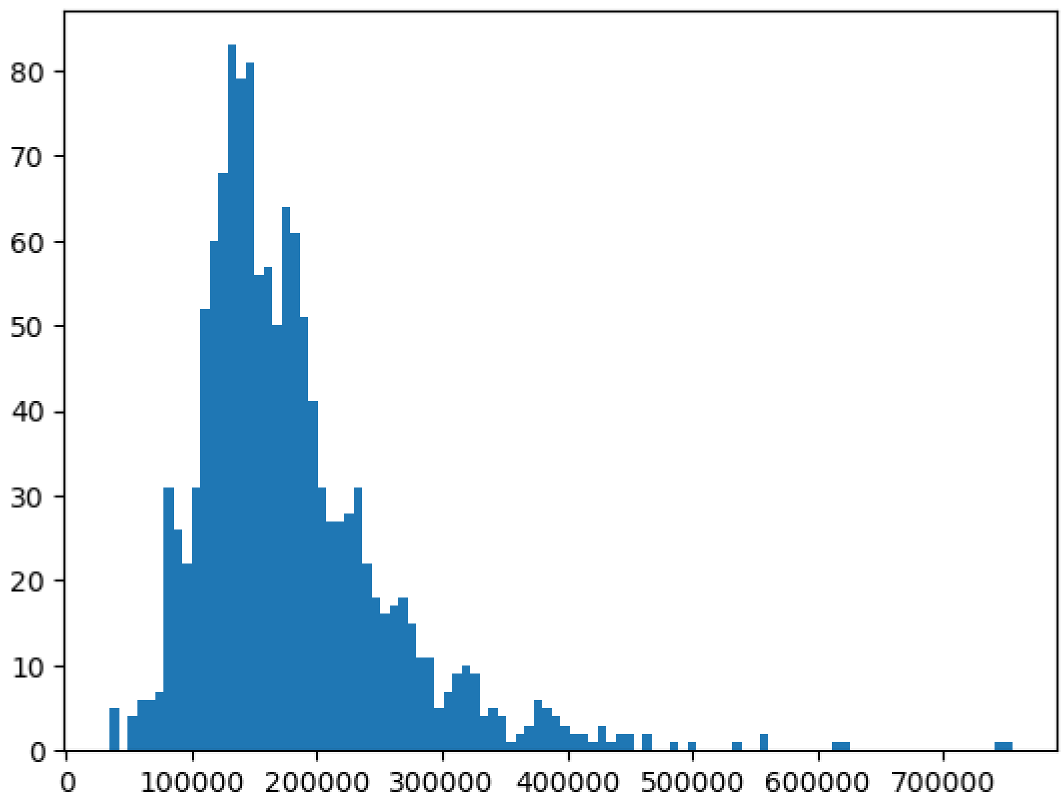
- Example 1: Truth: $50k, Prediction: \$100k
- Example 2: Truth: $500k, Prediction: \$550k
- RMSE: $50k
- MAPE: 45%

Model B

- Example 1: Truth: $50k, Prediction: \$60k
- Example 2: Truth: $500k, Prediction: \$600k
- RMSE: $71k
- MAPE: 20%

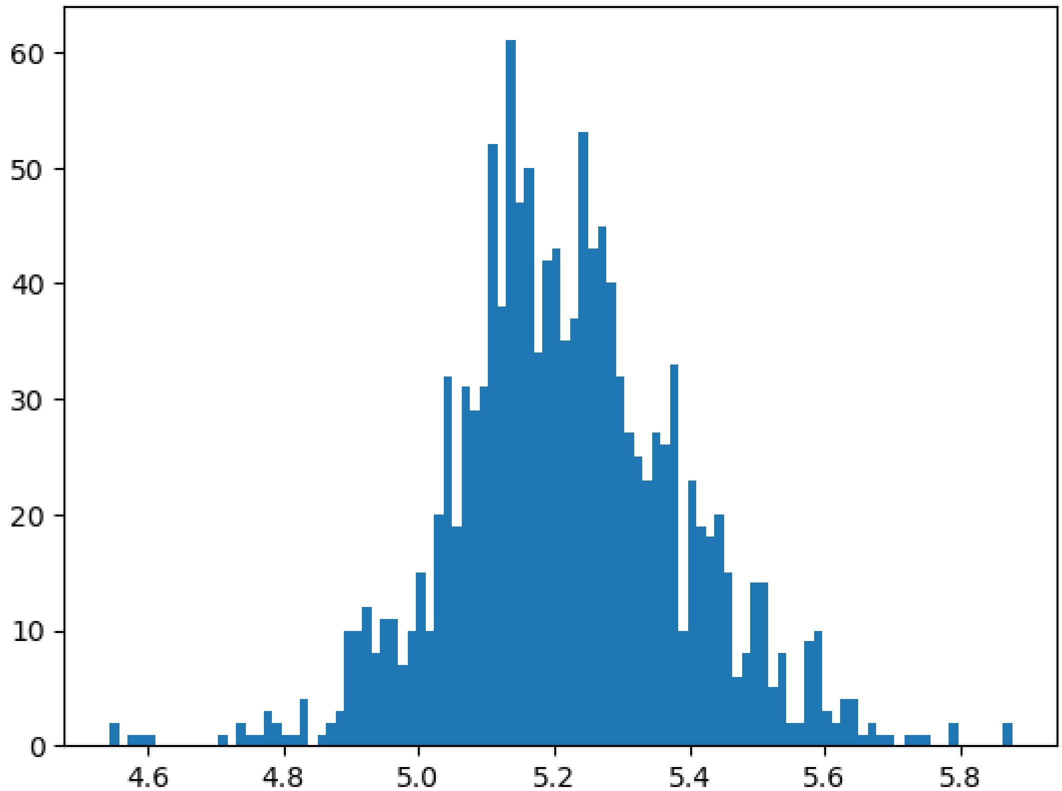
- How can we get `.fit()` to think about MAPE?
- A common practice which tends to work is log transforming the targets.
- That is, transform $y\rightarrow \log(y)$.

In [97]:
```
plt.hist(y_train, bins=100);
```



Processing math: 100%

In [98]:
```
plt.hist(np.log10(y_train), bins=100);
```



We can incorporate this in our pipeline using `sklearn`.

In [99]:
```
from sklearn.compose import TransformedTargetRegressor
```

In [100]:
```
ttr = TransformedTargetRegressor(
    Ridge(alpha=best_alpha), func=np.log1p, inverse_func=np.expm1
) # transformer for log transforming the target
ttr_pipe = make_pipeline(preprocessor, ttr)
```

In [101]:
```
ttr_pipe.fit(X_train, y_train); # y_train automatically transformed
```

In [102]:
```
ttr_pipe.predict(X_train)  # predictions automatically un-transformed
```

Out[102]:
```
array([221355.29528077, 170663.43286226, 182608.09768702, ...,
       248575.94877669, 132148.9047652 , 133262.17638244])
```

In [103]:
```
mape(y_test, ttr_pipe.predict(X_test))
```

Out[103]:
```
7.808600924240852
```

We reduced MAPE from ~10% to ~8% with this trick!

Processing math: 100%

Different scoring functions with `cross_validate`

- Let's try using MSE instead of the default R^2 score.

In [104]:
```
pd.DataFrame(
    cross_validate(
        lr_tuned,
        X_train,
        y_train,
        return_train_score=True,
        scoring="neg_mean_squared_error",
    )
)
```

Out[104]:

	fit_time	score_time	test_score	train_score
0	0.070120	0.027592	-7.060346e+08	-9.383069e+08
1	0.073525	0.027878	-1.239851e+09	-8.267971e+08
2	0.071608	0.048744	-1.125125e+09	-8.763019e+08
3	0.111845	0.030674	-9.819320e+08	-8.847908e+08
4	0.072874	0.026326	-2.268434e+09	-7.397199e+08

In [105]:
```
def mape(true, pred):
    return 100.0 * np.mean(np.abs((pred - true) / true))


# make a scorer function that we can pass into cross-validation
mape_scorer = make_scorer(mape, greater_is_better=True)

pd.DataFrame(
    cross_validate(
        lr_tuned, X_train, y_train, return_train_score=True, scoring=ma
    )
)
```

Out[105]:

	fit_time	score_time	test_score	train_score
0	0.077079	0.025021	9.699277	10.407124
1	0.076896	0.025295	10.803043	9.966190
2	0.094964	0.030100	11.836195	10.180734
3	0.090991	0.037569	10.784686	10.247198
4	0.081499	0.029051	12.196718	9.828607

Processing math: 100%

In [106]:

```
scoring = {
    "r2": "r2",
    "mape_scorer": mape_scorer,
    "neg_root_mean_square_error": "neg_root_mean_squared_error",
    "neg_mean_squared_error": "neg_mean_squared_error",
}

pd.DataFrame(
    cross_validate(lr_tuned, X_train, y_train, return_train_score=True,
).T
```

Out[106]:

	0	1	2	3	
fit_time	8.490086e-02	8.591008e-02	8.298492e-02	8.994293e-02	8.87
score_time	3.135419e-02	3.481293e-02	3.112602e-02	3.396201e-02	3.49
test_r2	8.668969e-01	8.200460e-01	8.262644e-01	8.511854e-01	6.10
train_r2	8.551369e-01	8.636241e-01	8.579735e-01	8.561893e-01	8.83
test_mape_scorer	9.699277e+00	1.080304e+01	1.183620e+01	1.078469e+01	1.21
train_mape_scorer	1.040712e+01	9.966190e+00	1.018073e+01	1.024720e+01	9.82
test_neg_root_mean_square_error	-2.657131e+04	-3.521152e+04	-3.354288e+04	-3.133579e+04	-4.76
train_neg_root_mean_square_error	-3.063179e+04	-2.875408e+04	-2.960240e+04	-2.974543e+04	-2.71
test_neg_mean_squared_error	-7.060346e+08	-1.239851e+09	-1.125125e+09	-9.819320e+08	-2.26
train_neg_mean_squared_error	-9.383069e+08	-8.267971e+08	-8.763019e+08	-8.847908e+08	-7.39

In [107]:

```
mape(y_test, lr_tuned.predict(X_test))
```

Out[107]: 9.496387589496008

Using regression metrics with `scikit-learn`

- In `sklearn` you will notice that it has negative version of the metrics above (e.g., `neg_mean_squared_error`, `neg_root_mean_squared_error`).
- The reason for this is that scores return a value to maximize, the higher the better.
- If you define your own scorer function and if you do not want this interpretation, you can set the `greater_is_better` parameter to False

Questions for class discussion

True/False

1. Price per square foot would be a good feature to add in our `X`.
2. The `alpha` hyperparameter of `Ridge` has similar interpretation of `C` hyperparameter of `LogisticRegression`; higher `alpha` means more complex model.

Processing math: 100%

3. In regression, one should use MAPE instead of MSE when relative (percent) error matters more than absolute error.
4. A lower RMSE value indicates a better model.
5. We can use still use precision and recall for regression problems but now we have other

Summary

- House prices dataset target is price, which is numeric -> regression rather than classification
- There are corresponding versions of all the tools we used:
  - `DummyClassifier -> DummyRegressor`
  - `LogisticRegression -> Ridge`
- `Ridge` hyperparameter `alpha` is like `LogisticRegression` hyperparameter `C` , but opposite meaning
- We'll avoid `LinearRegression` in this course.

- Scoring metrics
- R^2 is the default .score(), it is unitless, 0 is bad, 1 is best
- MSE (mean squared error) is in units of target squared, hard to interpret; 0 is best
- RMSE (root mean squared error) is in the same units as the target; 0 is best
- MAPE (mean average percent error) is unitless; 0 is best, 100 is bad

In [ ]:

```
 
```

Processing math: 100%